Experimental comparison of the quality of TFSM-based test suites for the UML diagrams

Rustam Galimullin
Department of Radiophysics
Tomsk State University
Tomsk, Russia
nihilkhaos@gmail.com

Abstract— **The paper presents the experimental comparison of the quality of three test suites based on the model of a Finite State Machine with timeouts, namely, the explicit enumeration of faulty mutants, transition tour and TFSM-based black-box test. Test suites are then applied to the program, automatically generated via the UML tool. The experimental results on the quality of the above mentioned test suites and the corresponding analysis are presented.**

Keywords—Finite State Machines with timeouts, the UML state machine diagrams, test suites

I. Introduction

Nowadays software failures of critical control systems are very expensive, and, thus, it is essential to provide high-quality testing at every stage of the system development. Many of such systems are formally described using the UML (the Unified Modeling Language) that has become the *de facto* standard for modeling software applications. The UML being a visual modeling language allows obtaining comprehensive and detailed information about a system under design, as well as provides a possibility for convenient update of the system. Correspondingly, the UML is widely used in software engineering, business project development, hardware design and in a number of other applications. The UML description can be automatically translated into a program code using proper tools and the developed software should be thoroughly tested. One of the formal models for testing UML-based software is a trace timed model. In this paper, we derive tests with the guaranteed fault coverage based on a timed Finite State Machine (FSM) augmented with timeouts [1], since FSMs are known to be an efficient model for deriving tests with the guaranteed fault coverage. The paper presents a case study for assessing the quality of test suites derived by three methods [2, 3, 4], which is estimated for the example of a phone line; the UML description of this project is taken from [5]. Using the tool *Visual Paradigm for UML* 8.0 [6] a JAVA code is generated for this application that serves a sample when assessing the test suite quality. We first check whether the initial program passes all the derived test suites. At the second step, some practical faults are injected into the initial program. Applying to each mutant tests, which were derived on the basis of timed FSM, we check whether injected faults can be detected with the test suites and analyze the reason when some faults cannot be detected with some/all derived test suites.

II. Preliminaries

The model we use, TFSM, is an extension of a classical FSM that is described as a finite set of states and transitions between them. Every transition is labeled by an *input/output* pair, where an input triggers the transition and an output is a system response to a given input. Formally, a *timed Finite State Machine* (TFSM) is a 6-tuple $S = (S, I, O, s_0, \lambda_S, \Delta_S)$ where S is a finite nonempty set of states with the initial state s_0, I and O are finite disjoint input and output alphabets, $\lambda_S \subseteq S \times I \times O \times S$ is transition relation and Δ_S: $S \rightarrow S \times (N \cup \infty)$ is a delay function defining timeout for each state [1]. If no input is applied at a current state during the appropriate time period (timeout), a TFSM can move to another prescribed state. A TFSM is called *deterministic* if for each pair $(s, i) \in S \times I$ there is at most one pair $(o, s') \in O \times S$ such that $(s, i, o, s') \in \lambda_S$, otherwise it is called *nondeterministic*. If for each pair $(s, i) \in S \times I$, there is at least one pair $(o, s') \in O \times S$ such that $(s,i, o, s') \in \lambda_S$ then S is said to be *complete*, otherwise it is *partial*.

A *timed input* symbol is a pair $\langle i, t\rangle \in I \times Z_0^+$, where Z_0^+ is a set of nonnegative integers. The timed input symbol shows that the input symbol i is applied at the moment when the value of the time variable is equal to t. A sequence of timed input symbols $\langle i_1, t_1\rangle \ldots \langle i_k, t_k\rangle$ is a *timed input sequence* of length k.

Let $S = (S, I, O, s_0, \lambda_S, \Delta_S)$ and $Q = (Q, I, O, q_0, \lambda_Q, \Delta_Q)$ be complete TFSMs. TFSMs S and Q are said to be *non-separable* if the sets of output responses of these TFSMs to any timed input sequence α intersect; i.e. $out_S (s_0, \alpha) \cap out_Q (q_0, \alpha) \neq \varnothing$. Otherwise, the TFSMs are *separable*. A timed input sequence α, such that $out_S (s_0, \alpha) \cap out_Q (q_0, \alpha) = \varnothing$ is called a *separating sequence* for TFSMs S and B. TFSM S is a *submachine* of TFSM Q if $S \subseteq Q$, $s_0 = q_0$ and each timed transition $(s, \langle i, t\rangle, o, s')$ of S is a timed transition of Q.

Intersection $S \cap Q$ of two FSMs is the largest submachine of $P = (P, I, O, p_0, \lambda_P, \Delta_P)$, where $P = S \times K \times Q \times K$, $K = \{0, \ldots, k\}$, $k = min(max\ \Delta_S(s),\ max\ \Delta_Q(q))$, the initial state is

quadruple (s_0, 0, p_0, 0). Transition relation λ_P and function Δ_P are defined by the following rules:

1. The transition relation λ_P contains quadruple $[(s, k_1, q, k_2), i, o, (s', 0, q', 0)]$ iff $(s, i, o, s') \in \lambda_S$ and $(q, i, o, q') \in \lambda_Q$.

2. Time function is defined as $\Delta_P(s, k_1, q, k_2) = [(s, k'_1, q, k'_2), k]$, $k = \min(\Delta_S(s)\downarrow_N - k_1,\ \Delta_Q(q)\downarrow_N - k_2)$. State $(s', k_1', q', k_2') = (\Delta_S(s)\downarrow_S, 0, \Delta_Q(q)\downarrow_Q, 0)$, if $\Delta_S(s)\downarrow_N = \infty$, $\Delta_Q(q)\downarrow_N = \infty$ or $(\Delta_S(s)\downarrow_N - k_1) = (\Delta_Q(q)\downarrow_N - k_2)$. If $(\Delta_S(s)\downarrow_N - k_1)$, $(\Delta_Q(q)\downarrow_N - k_2) \in Z_+$ и $(\Delta_S(s)\downarrow_N - k_1) < (\Delta_Q(q)\downarrow_N - k_2)$, then state $(s', k_1', q', k_2') = (\Delta_S(s)\downarrow_S, 0, q, k_2 + k)$. If $(\Delta_S(s)\downarrow_N - k_1)$, $(\Delta_Q(q)\downarrow_N - k_2) \in Z_+$ and $(\Delta_S(s)\downarrow_N - k_1) > (\Delta_Q(q)\downarrow_N - k_2)$, then state $(s', k_1', q', k_2') = (s, k_1 + k, \Delta_Q(q)\downarrow_Q, 0)$.

III. CASE STUDY

As a running example, we consider a simple phone line state machine diagram, taken from [5].

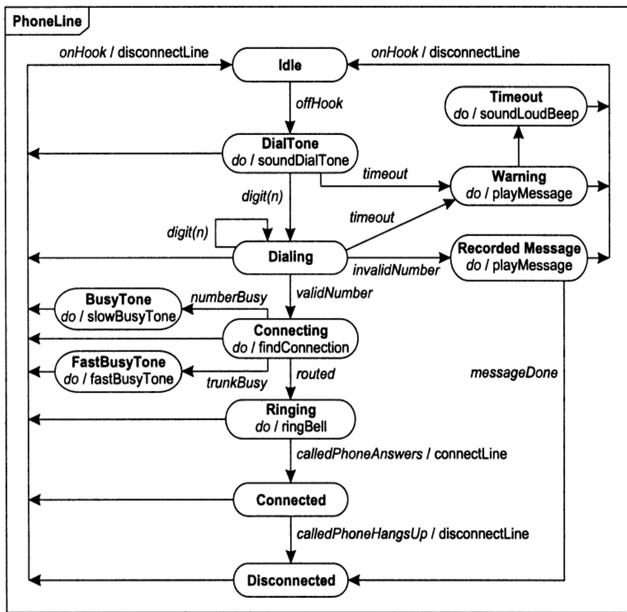

Fig. 1. Phone line state machine diagram

When the device is at *Idle* state it is possible to pick up the phone (*offHook*), and to get *soundDialTone* as an output. The state diagram is at *Dialing* state when a user enters the number (*digit* (*n*)). If the number cannot be served (*invalidNumber*), the corresponding message is played (*playMessage*). Otherwise, the device enters the state *Connecting*. At this state, four different events are possible. Either the number or a trunk is busy, and in this case, the user should hang up, or the phone will connect (*routed*). After the connection there is the ring (*ringBell*) and, finally, the conversation takes place (state *Connected*). After the conversation, either the user or her/his partner hangs up. In both cases, the line will be disconnected. With a case tool *Visual Paradigm for UML* 8.0 JAVA program code of this diagram was automatically generated.

A TFSM that describes such a state machine diagram is in Figure 2.

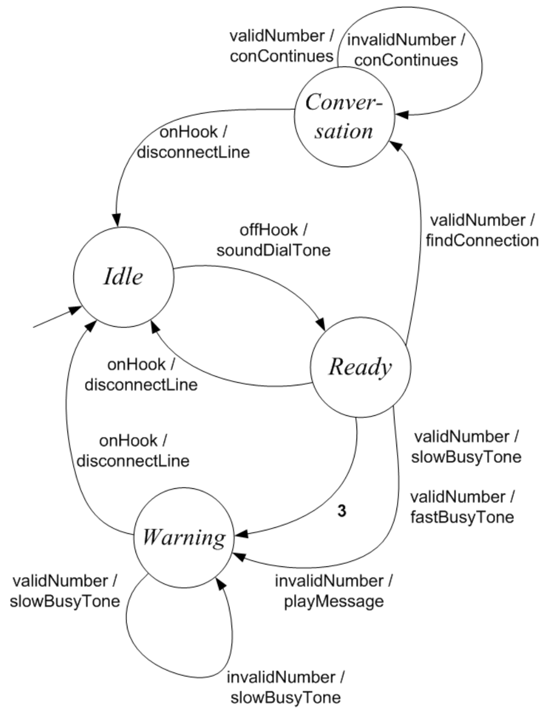

Fig. 2. TFSM that describes the phone line, presented in Fig. 1.

The TFSM has four states and one timeout at state *Ready*. The initial state is *Idle*. If the user picks up the phone (*offHook*), a dial tone is played (*soundDialTone*), and the TFSM changes its state to *Ready*. If the user does not interact with the system for a certain period of time, 3 time units for instance, the state will be spontaneously changed to *Warning*. At the *Ready* state, a user also can hang up the phone (*onHook*) and in this case, the line will be disconnected (*disconnectLine*). If the user enters a valid number (*validNumber*), the TFSM can response in three different ways. The first option is a busy number (*slowBusyTone*) and in this case, the system changes its state to *Warning*. The second option is a busy trunk (*fastBusyTone*). The last option is that a conversation starts. In other words, the corresponding TFSM is nondeterministic. In the *Warning* state none of entered numbers (*validNumber* and *invalidNumber*) affects the system. Being in this state the user can only hang up (*onHook*) and the same situation occurs in the *Conversation* state.

Here we notice that a TFSM is also partial and cannot be augmented to a complete TFSM as it is usually done when deriving tests against a partial specification FSM. The reason is that after *onHook* input we cannot apply the same input. This input can be applied only after the input *offHook*.

IV. METHODS OF TEST DERIVATION

Three TFSM-based methods for the test suite derivation are considered in the paper. The first method (Method 1) is based on the explicit enumeration of faulty mutants. Given the specification TFSM, some faults are injected in it, i.e. some mutant TFSMs are constructed, and for each mutant an input sequence that separates the specification TFSM and this mutant is derived [2]. A separating sequence is an input sequence such that the sets of output responses of two TFSMs to this sequence do not intersect, and since the TFSMs can be nondeterministic we use a separating sequence instead of traditional distinguishing sequence [7]. In order to derive a separating sequence we first construct the intersection of two given TFSMs and then a truncated successor tree is

constructed for the intersection. In the paper, we consider only six mutants which describe meaningful faults for our running example.

1. A fault related to the timeout at state *Ready*. In this case, the TFSM has a transition, labelled with timeout 4, instead of 3. For this pair of FSMs, specification (Fig. 2.) and mutant TFSMs, a separating sequence is (⟨*offHook*, 0⟩, ⟨*validNumber*, 3⟩).
2. Another wrong timeout. But now it is smaller (*e.g.* 1) than that of the specification TFSM. By direct inspection, one can assure, that for this mutant a separating sequence is (⟨*offHook*, 0⟩, ⟨*validNumber*, 2⟩).
3. The situation when having an invalid number as an input the connection is still found. For this particular case, a separating sequence is (⟨*offHook*, 0⟩, ⟨*invalidNumber*, 0⟩).
4. The situation when during the conversation a user accidentally types some digits (a number), and the slow busy tone is played. In this case, an input sequence (⟨*offHook*, 0⟩, ⟨*validNumber*, 0⟩, ⟨*validNumber*, 0⟩) is a separating sequence.
5. The situation when being at state "*Warning*" we can make a call anyway. For this case, a separating sequence is (⟨*offHook*, 0⟩, ⟨*validNumber*, 3⟩).
6. The situation when conversation is impossible (*i.e.* there is no transition to state *Conversation*) and in this case, a separating sequence is (⟨*offHook*, 0⟩, ⟨*validNumber*, 0⟩).

We consider the set of the above mentioned separating sequences as a test suite for explicit enumeration of mutants. Thus, TS_1 = {(⟨*offHook*, 0⟩, ⟨*validNumber*, 3⟩), (⟨*offHook*, 0⟩, ⟨*validNumber*, 2⟩), (⟨*offHook*, 0⟩, ⟨*invalidNumber*, 0⟩), (⟨*offHook*, 0⟩, ⟨*validNumber*, 0⟩, ⟨*validNumber*, 0⟩)}.

The second method (Method 2) for deriving a test suite against TFSMs with the guaranteed fault coverage is based on the correlation between TFSM and FSM (Procedure 1) [4]. To transform a timed FSM into a classical FSM we add a special input symbol 1 that corresponds to the notion of *waiting one time unit*, and a special output – *N* that corresponds to the case when there is no reply from the machine. If at state *s* a timeout value *T* is greater than 1, then we add $(T - 1)$ copies of state *s* with corresponding outgoing transitions. If TFSM have *n* states and the maximum finite timeout is T_{max}, the corresponding FSM may have up to $n \cdot T_{max}$ states. In Figure 3, there is an FSM that corresponds to the specification TFSM in Figure 2.

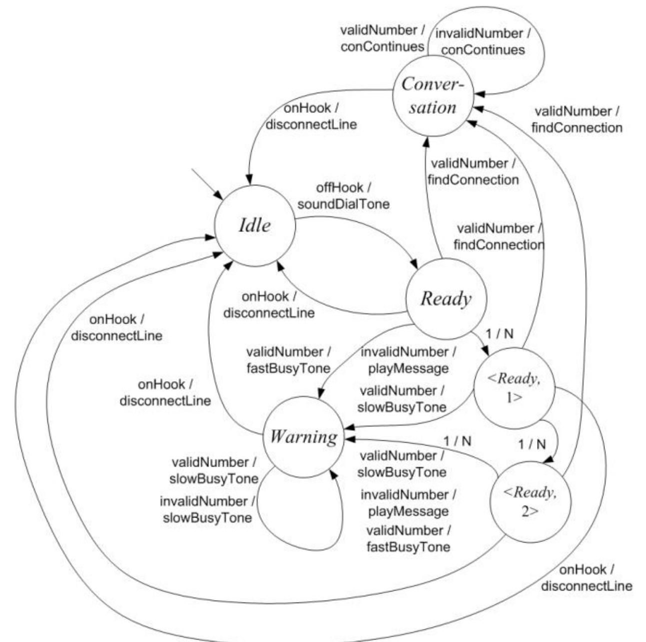

Fig. 3. FSM that corresponds to the TFSM, presented in Fig. 2.

Given a classical FSM, a test suite that is complete w.r.t. to output faults can be derived as a transition tour of the FSM [3]. A transition tour of an FSM is a finite set of input sequences which started at the initial state traverse each FSM transition. A corresponding transition tour can be derived for an FSM that is derived from corresponding TFSM.

Proposition 1. Given a test suite *TS* for TFSM based on a transition tour of an FSM output by Procedure 1, the *TS* detects each output fault of the TFSM.

A transition tour for the FSM (Fig. 3) is a test suite TS_2 = {(⟨*offHook*, 0⟩, ⟨*validNumber*, 1⟩, ⟨*validNumber*, 0⟩, ⟨*onHook*, 0⟩), (⟨*offHook*, 0⟩, ⟨*validNumber*, 1⟩, ⟨*validNumber*, 0⟩, ⟨*onHook*, 0⟩), (⟨*offHook*, 0⟩, ⟨*validNumber*, 0⟩), (⟨*offHook*, 0⟩, ⟨*invalidNumber*, 2⟩), (⟨*offHook*, 0⟩, ⟨*validNumber*, 2⟩), (⟨*offHook*, 0⟩, ⟨*onHook*, 1⟩), (⟨*offHook*, 0⟩, ⟨*offHook*, 2⟩), (⟨*offHook*, 0⟩, ⟨*onHook*, 0⟩), (⟨*offHook*, 0⟩, ⟨*invalidNumber*, 0⟩)}.

Consider now the third test derivation method proposed in the paper [4]. The method has two testing assumptions: the upper bound on the number of states of a TFSM under test (implementation under test, IUT) and the largest finite timeout at a state of the IUT are known. Authors show that in this case, a complete test suite obtained directly from a given TFSM is much shorter than a complete test suite that is derived based on a corresponding FSM by the use of corresponding FSM based methods [8]. The procedure for test derivation consists of three steps. We first identify each state of the specification TFSM using separating sequences. At the next step, we check all transitions at each state, i.e. reach a state, execute a transition and execute corresponding separating sequences. At the last step, timeouts are tested: for this purpose at each state we apply inputs $(i, 1), \ldots, (i, T + 1)$ when *T* is the largest timeout of the IUT. The method was proposed for reduced complete deterministic TFSMs; however, we use it also for nondeterministic partial TFSMs adding separating sequences after each transition. We also assume that if a timeout at a state of the specification TFSM is ∞ then the IUT has the same timeout. Correspondingly, for the specification TFSM (Fig. 2.) we do not check the initial state *Idle*; all other states can be identified by separating sequences listed below. For

state *Warning* we have a separating sequence (⟨*offHook*, 0⟩, ⟨*invalidNumber*, 0⟩, ⟨*invalidNumber*, 0⟩), for state *Conversation* - (⟨*offHook*, 0⟩, ⟨*validNumber*, 0⟩, ⟨*invalidNumber*, 0⟩) and for state *Ready* - (⟨*offHook*, 0⟩, ⟨*onHook*, 0⟩). At the second step all the transitions are checked. We use a transition tour {(⟨*offHook*, 0⟩, ⟨*invalidNumber*, 0⟩, ⟨*invalidNumber*, 0⟩, ⟨*onHook*, 0⟩), (⟨*offHook*, 0⟩, ⟨*validNumber*, 0⟩, ⟨*invalidNumber*, 0⟩, ⟨*onHook*, 0⟩), (⟨*offHook*, 0⟩, ⟨*onHook*, 0⟩)}, where each sequence is augmented with a corresponding separating sequence At the final step timeouts are checked and we derive the following sequences: {(⟨*offHook*, 0⟩, ⟨*invalidNumber*, 1⟩), (⟨*offHook*, 0⟩, ⟨*invalidNumber*, 2⟩), (⟨*offHook*, 0⟩, ⟨*invalidNumber*, 3⟩)}. Thus for given FSM test suite is TS_3 = {(⟨*offHook*, 0⟩, ⟨*invalidNumber*, 0⟩, ⟨*invalidNumber*, 0⟩), (⟨*offHook*, 0⟩, ⟨*validNumber*, 0⟩, ⟨*invalidNumber*, 0⟩), (⟨*offHook*, 0⟩, ⟨*onHook*, 0⟩), (⟨*offHook*, 0⟩, ⟨*invalidNumber*, 0⟩, ⟨*invalidNumber*, 0⟩, ⟨*onHook*, 0⟩), (⟨*offHook*, 0⟩, ⟨*validNumber*, 0⟩, ⟨*invalidNumber*, 0⟩, ⟨*onHook*, 0⟩), (⟨*offHook*, 0⟩, ⟨*invalidNumber*, 1⟩), (⟨*offHook*, 0⟩, ⟨*invalidNumber*, 2⟩), (⟨*offHook*, 0⟩, ⟨*invalidNumber*, 3⟩)}.

V. EXPERIMENTAL RESULTS

We now consider the set of possible program faults which is listed below.

1. The transition from state *Ready* to state *Conversation* is triggered by input *validNumber*, but the output is *fastBusyTone* instead of *findConnection*.
2. The timeout at state *Ready* is greater than that in the specification TFSM, e.g. timeout equals six instead of three.
3. There is a new transition from state *Ready* to state *Warning* under input *onHook* with *fastBusyTone* output.
4. A fault is inside the program. While scanning the valid number set there is a *while loop*, and if an input number coincides with one in the list, then Boolean variable *flag* is true, otherwise – false.

```
while ((strLine = br.readLine()) != null)
{
        if (s == null ? strLine == null :
                s.equals(strLine))
        {
                flag = true;
        }
}
```

   The fault is as follows. If an entered number *is not* in the list, then *flag* is still true and in order to get the mutant we add **!** in an *if clause*.

```
while ((strLine = br.readLine()) != null)
{
        if (s == null ? strLine == null :
                !s.equals(strLine))
        {
                flag = true;
        }
}
```

   The fault implies that all entered numbers are valid.

5. A new state is added. From state *Ready* it is possible to enter a new *Wait* state via *validNumber* input. This means that having a valid number as an input a user would listen to a special message, e.g. "Connection is set up. Please wait". This is modeled by an output *findConnection*. On input *validNimber* in state *Wait* there is an output *convContinues*. If an *invalidNumber* is entered the implementation TFSM changes its state to state *Warning* with *fastBusyTone* output. Finally, if *onHook* input is applied the machine is at *Idle* state and *disconnectLine* output is produced.
6. There is a new timed transition from state *Conversation* to state *Ready* after 8 time units. This means, that after 8 time units the conversation is automatically finished.

We first apply each test case to the initial program to be sure that the program produces expected output sequences to every test case. Then all the above faults were injected into the initial program. For each test suite, each test case was applied to a mutant program. A fault was *detected* by a test suite when there was at least one test case of the test suite such that the output responses of the initial program and of a mutant program were different. The results are presented in Table I, where '+' means that this fault can be detected by a corresponding test suite.

TABLE I. EXPERIMENTAL RESULTS

	1	2	3	4	5	6
TS_1	+	+	-	+	-	-
TS_2	+	+	+	+	-	-
TS_3	+	+	+	+	+	-

As we can see, Faults 3, 5 and 6 were not found by TS_1; the reason can be that we did not consider corresponding mutants for the specification TFSM. Despite of the fact, that some of such mutant program still can be detected in this case, we were 'unlucky'. Test suite TS_2 did not detect Faults 5 and 6, since when considering a transition tour, we assume that the number of states of an IUT is the same as of the specification TFSM. Finally, Fault 6 was not detected even by TS_3 because we also violated testing hypothesis about an implementation TFSM. Nevertheless, we could conclude that a transition tour where each sequence is appended with corresponding separating sequence can detect more faults and thus, such augmentation is worth for improving the quality of a generated test suite.

V. CONCLUSIONS

In this paper, we considered three methods for the deriving tests based on the model of an FSM augmented with input and output timeouts for automatically generated program code of an UML project. Using a simple running example we illustrate that a transition tour of the specification TFSM augmented with corresponding separating sequences is a test suite of a good quality and this test suite detects not only faults it is derived for, but also other faults, including those which increase the number of states of an implementation TFSM.

ACKNOWLEDGMENT

I would like to express my sincere gratitude to my scientific supervisor professor Nina Yevtushenko for her invaluable support during the work on paper.

REFERENCES

[1] M. Gromov, D. Popov and N. Yevtushenko, "Deriving test suites for timed Finite State Machines," Proc. of IEEE East-West Design & Test Symposium, pp. 339-343, 2008.

[2] N. Shabaldina and R. Galimullin, "On deriving test suites for nondeterministic Finite State Machines with time-outs," Programming and computer science, vol. 38, pp. 127-133, 2012.

[3] M. Zhigulin "TFSM-based methods of fault detection tests synthesis with guaranteed fault coverage for discrete controlling systems", PhD thesis, TSU, Tomsk, 2012. (in Russian)

[4] M. Zhigulin, S. Maag, A. Cavalli and N. Yevtushenko, "FSM-based test derivation strategies for systems with time-outs," Proc. of the 11th conference on quality software (QSIC), pp. 141-149, 2011.

[5] J. E. Rumbaugh and M.R. Blaha "Object-oriented modeling and design with UML (2 ed.)," Pearson Education, 2005.

[6] Visual Paradigm [Electronic resource] - http://www.visual-paradigm.com/

[7] A. Gill, "Introduction to theory of Finite State Machines," McGraw-Hill, 1962.

[8] R. Dorofeeva, K. El-Fakih, S. Maag, A. Cavalli, N. Yevtushenko (2010) "FSM-based Conformance Testing Methods: A Survey Annotated with Experimental Evaluation," Information and Software Technology, Elsevier, 52, pp. 1286-1297, 2010.